MTKPowerConverters.jl: Converter Modeling and Simulation in the Julia SciML Ecosystem

Sergio A. Dorado-Rojas†, Nicolas Dimate†, Patrick Panciatici, Sorin Olaru, Daniel K. Molzahn

***Abstract*—`MTKPowerConverters.jl` is a proof-of-concept open-source Julia library for the simulation of power electronic converters. The library adopts the design philosophy of the Modelica language: acausal, equation-based components compose without rewriting, and physical model specification stays cleanly separated from the numerical solver. Built on `ModelingToolkit.jl` and relying on the solver catalog of `DifferentialEquations.jl`, the models are natively embedded in the Julia SciML ecosystem, potentially enabling automatic differentiation and physics-informed analysis workflows unavailable in closed tools. Three experiments validate the concept. Smooth surrogates of the PWM carrier waveforms are compared against their ideal discontinuous counterpart to quantify the accuracy versus solve-time trade-off. An open-loop buck converter switching at $100\,\mathrm{kHz}$ is cross-validated against MATLAB/Simscape, with close steady-state agreement and a smaller end-to-end solve time. Finally, an open-loop single-phase inverter is benchmarked against the same reference under numerically demanding conditions, confirming that the approach is robust and comparable in accuracy to commercial tools.**

I. Introduction

Modern power electronics simulation tools are highly effective at producing results but deliberately opaque about how they do so. Classical circuit simulators (SPICE, Saber) hard-wire their equation structure into the underlying algorithms, restricting symbolic analysis to topology parsing and therefore preventing seamless coupling with external physical domains [1]. Other environments, such as MATLAB/Simscape and PLECS, compile component models into proprietary binaries, which makes their equations not open for inspection [2].

This opacity has documented consequences. Cellier and Kofman [1, §1.1] observe that a user who understands modeling but treats the simulation environment as a black box is left with no principled mechanism for debugging when numerical results deviate from expectations. From a scientific reproducibility standpoint, Zeigler et al. [3] argue that opaque tools prevent the exchange of submodels for hypothesis testing, thereby forcing researchers to rebuild and reverify entire models from scratch.

The equation-based modeling paradigm, pioneered by the Modelica language [4], [5], resolves this by making the mathematical model the user interface rather than an implementation detail [1]. Transparency follows from Modelica's design. A Modelica tool needs the model equations in symbolic form to perform index reduction [4], and in open-source libraries the same equations the compiler processes remain readable to the modeler.

`MTKPowerConverters.jl`[1] applies these principles to power electronics. It constitutes a proof of concept of a Julia library of acausal, equation-based converter components built on `ModelingToolkit.jl` [6]. Each component lives in a Julia source file containing its governing equations in readable form. Since the models are built natively within the Julia SciML ecosystem [7], the library inherits all of its functionalities, such as automatic differentiation and physics-informed workflows, among others.

To the best of the authors' knowledge, the closest related Julia effort at the time of writing is `ElectricGrid.jl` [8], an open-source library for three-phase microgrid simulation with classical and reinforcement-learning control. `ElectricGrid.jl` operates at the controller sampling rate (10–50 kHz) and represents each converter as a linear time-invariant state-space model with modulation indices as inputs, deliberately abstracting switching dynamics into averaged voltages. `MTKPowerConverters.jl` addresses the complementary abstraction level by resolving individual switching events at the carrier frequency. It makes every circuit equation directly visible, filling a gap that grid-level tools built on averaged converter models leave by design.

The remainder of this paper is organized as follows. Section II revisits the `ModelingToolkit.jl` design workflow and the acausal equation-based modeling approach. Section III introduces the `MTKPowerConverters.jl` structure and its dependencies, with particular emphasis on the smooth signal approximation at the core of the switching simulation. Section IV presents the two benchmark converters, an open-loop buck converter and a single-phase inverter, and their model assembly. Section V presents the numerical experiments and their analysis, covering the comparison of pulse width modulation (PWM) approximation strategies and the cross-validation against MATLAB/Simscape. Section VI discusses potential analysis workflows enabled by the native

† denotes equal contribution.

This work has been funded in part by the Chateaubriand Fellowship of the Embassy of France in the United States, and in part by the National Science Foundation under grant number 2112533: NSF Artificial Intelligence Research Institute for Advances in Optimization (AI4OPT).

N. Dimate is an M.Sc. Graduate from ENSTA Paris, Institut Polytechnique de Paris, Paris, France. Email: `junmorenodi@gmail.com`

S. A. Dorado-Rojas and D. K. Molzahn are with the School of Electrical and Computer Engineering at Georgia Institute of Technology, Atlanta, GA 30313, USA. Email: {`sadr, molzahn`}`@gatech.edu`.

S. Olaru and P. Panciatici are with the Laboratoire des Signaux et Systèmes (L2S) and the RTE Chair, CentraleSupélec, Université Paris-Saclay, 35510 Gif-sur-Yvette, France. Emails: {`sorin.olaru, patrick.panciatici`}`@centralesupelec.fr`.

[1]Source code: https://github.com/sergio-dorado/MTKPowerConverters.jl.

integration with the SciML ecosystem. Finally, Section VII concludes this work and outlines future development.

II. Background: Acausal Equation-Based Modeling

A. ModelingToolkit.jl Workflow

`ModelingToolkit.jl` is a Julia package for equation-based modeling, enabling object-oriented modeling of engineering systems [6]. Through `ModelingToolkit.jl`, it is possible to construct modularly symbolic instances of mathematical "systems" such as `NonlinearSystem` for nonlinear algebraic equations, `ODESystem` for nonlinear ordinary differential equations and differential algebraic equation systems, or `OptimizationSystem` for nonlinear mathematical programming problems.

For power electronics, `ModelingToolkit.jl` enables to build circuit elements by extending the electrical building blocks of `ModelingToolkitStandardLibrary.jl`, which are themselves defined by current and voltage relations.

Electrical Interfaces: A `Pin` is the elementary connector of electrical components. Each pin is associated with a voltage $v(t)$ and a current $i(t)$ value. When two pins ⓐ and ⓑ are connected, they satisfy the connection constraints, $v_a(t) = v_b(t)$ and $i_a(t)+i_b(t) = 0$, which correspond to equal potential and conservation of current at the connection node. These equations provide a local basis from which Kirchhoff's laws emerge at the circuit level.

A `OnePort` is not a circuit element in itself but an interface for two-terminal elements. It provides two pins, ⓟ and ⓝ by convention, the port variables $v(t) = v_p(t) - v_n(t)$ and $i(t) = i_p(t)$, and the port constraint $i_p(t) + i_n(t) = 0$. A concrete element extends this interface by adding its constitutive relation $\boldsymbol{f}(t; v(t), i(t), \boldsymbol{\theta}) = \mathbf{0}$, where $\boldsymbol{\theta}$ collects its parameters. Resistors, capacitors, inductors, and the two-terminal semiconductor models in this work all share this interface.

In the implementation, each component is a function that returns a `System`, and a circuit is assembled from multiple such objects, each comprising a set of symbolic equations, parameters, and observed variables, connected through pins. To simulate a circuit model, the compiler assembles the connection equations across the network (Kirchhoff's Laws), performs Pantelides index reduction [1, §8], and generates a minimal reduced-index system of equations suitable for standard solvers, without requiring user intervention. The result is a causal model, in which every equation has been assigned the variable it is solved for.

This pipeline mirrors the Modelica compilation chain [4]: block lower triangular transformation, index reduction, and code generation produce a numerically integrable ordinary differential equation (ODE) from a declarative equation specification. Cellier and Kofman [1] frame this as the clean separation of the mathematical model (the user interface, optimized for human comprehension) from the simulation program (the run-time interface, optimized for numerical efficiency). `MTKPowerConverters.jl` inherits this separation from `ModelingToolkit.jl`, so every component source file contains only its governing equations, and the compiler handles the rest.

Component Example: The `PiecewiseLinearDiode` model in Snippet I illustrates how an element is defined using `ModelingToolkit.jl` and its standard library electrical interfaces. For this particular circuit element, the constitutive relation corresponds to

$$i(t;\boldsymbol{\theta}) = \begin{cases} G_{\text{off}}\, v(t), & v(t) < V_{\text{knee}}, \\ \dfrac{v(t) - V_{\text{knee}}}{R_{\text{on}}} + G_{\text{off}}\, V_{\text{knee}}, & v(t) \geq V_{\text{knee}}, \end{cases} \tag{1}$$

with model parameters $\boldsymbol{\theta} = (G_{\text{off}}, R_{\text{on}}, V_{\text{knee}})$. The two parameters R_{on} and G_{off} capture conduction-loss and leakage behavior respectively, and the offset $G_{\text{off}}V_{\text{knee}}$ in the second branch makes $i(t)$ continuous at $v(t) = V_{\text{knee}}$, so that no current jump is introduced at the switching boundary. If the simulated diode waveforms look unexpected, the user can read (1) directly, identify the parameter that governs the behavior, and adjust it, without any reverse engineering.

SNIPPET I
PiecewiseLinearDiode model implementation (excerpt).

```
@component function PiecewiseLinearDiode(; name,
    V_knee = 1e-3, R_on = 0.3, G_off = 1e-8)
    # ---------------------------------
    # INTERFACE EXTENSION
    # ---------------------------------
    @named oneport = OnePort(; v=0.0, i=0.0)
    @unpack v, i = oneport

    # ---------------------------------
    # PARAMETERS
    # ---------------------------------
    pars = @parameters begin
        R_on   = R_on
        G_off  = G_off
        V_knee = V_knee
    end

    # ---------------------------------
    # VARIABLES
    # ---------------------------------
    vars = @variables begin
        p(t) = 0.0
    end

    # ---------------------------------
    # EQUATIONS
    # ---------------------------------
    eqs = Equation[
        i ~ MTKBase.ifelse(v < V_knee, G_off * v, (v
            - V_knee) / R_on + G_off * V_knee)
    ]

    # ---------------------------------
    # OBSERVED
    # ---------------------------------
    obs = Equation[
        p ~ v * i
    ]

    return extend(System(eqs, t, [], pars; name,
        observed = obs), oneport)
end
```

The particular model in Snippet I instantiates the `OnePort` interface, which provides the port voltage $v(t)$ and current $i(t)$. The port current is a `flow` variable, which is what lets the compiler generate the conservation constraint at every node where several components meet, without the user writing it.

Note how the parameters $\boldsymbol{\theta}$ are declared separately from the model variables v and i, which are inherited from the `OnePort` interface. The model also defines observed quantities: derived variables whose values are determined from the model solution and can therefore be evaluated after the simulation without introducing additional governing equations. In this particular case, a power observable variable $p(t) = v(t)\, i(t)$ is defined without modifying the underlying equations or the component interface.

B. Acausality and Component Reuse

The implementation of the `PiecewiseLinearDiode` previously illustrated corresponds to a voltage-controlled element; i.e., given the voltage, the current is straightforwardly computed from the equation that depends on the diode's operating region. Consider, however, the case in which the current through the diode is imposed by the surrounding circuit. In the simulation jargon, the causality of the model $i(t) = h(v(t))$ would have to be inverted, i.e., a conditional statement would have to be appended to the model declaration to specify the component's behavior in such cases where $v(t) = g(i(t))$. This step is not necessary in `ModelingToolkit.jl`. Inspired by the Modelica language, the "context" of the simulation provides the necessary information to determine the component's causality, so this requirement is removed from the user during model specification. This is why Modelica and `ModelingToolkit.jl` are known as acausal modeling frameworks.

This is a powerful feature that allows the user to reuse the same component across different contexts without rewriting it. For instance, the same `PiecewiseLinearDiode` model serves as the freewheeling diode of a buck converter, where the inductor imposes the current and the constitutive relation is solved for the voltage, and as a rectifier diode placed between a voltage source and a filter capacitor, where the voltage is imposed and the relation is evaluated for the current. The same equations apply in both cases, and the compiler automatically determines the appropriate causality from the context in which the component is used.

III. Library Architecture

A. MTKPowerConverters.jl Package Organization

`MTKPowerConverters.jl` is organized into subsystems[2] with a clear dependency hierarchy, as presented in Fig. 1. In the present version, four subsystems are defined: `Signals`, `Components`, `Utils`, and `Sensors`. `Components` depends on `Signals`, `Utils`, and `Sensors`. The `Utils` and `Sensors` subsystems provide auxiliary capabilities for simulation and data processing.

[2]The term subsystem is used in the package-diagram sense of Fig. 1, that is, a named container of related `modules`, models, and functions. It does not denote an abstract type in the object-oriented sense.

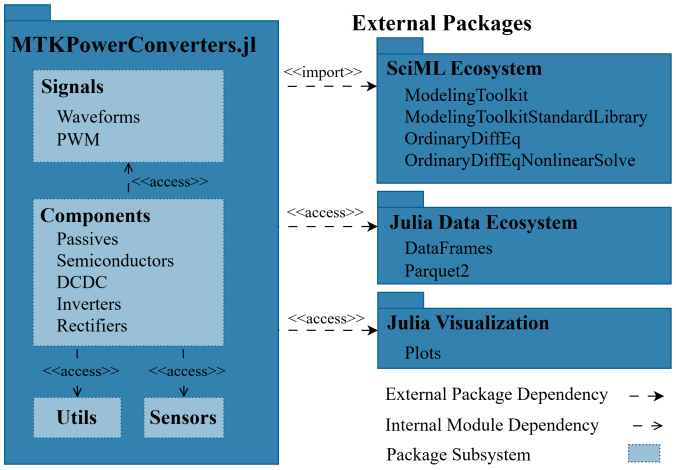


Fig. 1. High-level overview of the `MTKPowerConverters.jl` package.

All subsystems rely on the SciML ecosystem. `ModelingToolkit.jl` and `ModelingToolkitStandardLibrary.jl` provide the symbolic modeling framework and the electrical interfaces, while `OrdinaryDiffEq` and `OrdinaryDiffEqNonlinearSolve` provide the solvers for the resulting numerical problems. Other packages, such as `DataFrames`, `Parquet2`, and `Plots`, are used in a post-simulation phase to facilitate user exploration of results.

Signals: Switching is challenging for numerical simulation because it introduces discontinuities into the system dynamics [1]. The `Signals` subsystem therefore provides smooth models of PWM carriers and of the resulting switching signals that drive the converter switches, as documented in Section III-B.

Components: The `Components` subsystem contains specialized modules developed for building power converter models. The `Semiconductors` module contains the semiconductor models necessary for device-layer power converter modeling, e.g., the `PiecewiseLinearDiode` in Snippet I. The `Passives` module contains extended versions of the standard passive components of `ModelingToolkitStandardLibrary.jl`, such as variable resistors and non-ideal capacitors and inductors. The `DCDC`, `Inverters`, and `Rectifiers` modules provide predefined DC-DC, DC-AC, and AC-DC converter models, respectively, to facilitate integration into larger systems.

B. Smooth PWM Signal Waveform Generation

All numerical ODE solution methods, including the stiff implicit methods widely employed in power electronics simulation software, approximate solution trajectories using Taylor series expansions about the current time, which requires local differentiability of the embedded functions in the model equations.

An inherent difficulty of switching dynamic simulations is that the switching behavior introduces discontinuities into the system dynamics, which can be addressed by the solver at the cost of additional computational overhead[3].

For switching power converters, the conventional workarounds for these computational challenges in simulations are: (i) averaged switch models, which replace the high-frequency switching waveforms with their moving-average equivalents, removing high-frequency content at the cost of fidelity [2], [9]; and (ii) hardware oversampling with interpolation-extrapolation compensation, used in fixed-step real-time simulators such as Typhoon HIL to locate switching instants [10]. Neither approach suits the variable-step ODE solvers on which equation-based tools rely.

`MTKPowerConverters.jl` takes a third route, complementary to the event handling of the compiler. To ease the simulation burden, the discontinuous waveforms used in switching converter models, such as the steps, ramps, and sawtooths of PWM generators, are replaced by smooth surrogates with a controllable transition width, so that fewer discontinuities reach the solver. Those that remain, in the piecewise constitutive relations of switches and diodes, are turned into discrete events by the `IfLifting` pass of `mtkcompile` (Section IV-A).

Step Functions: Every waveform in the `Signals` subsystem is built from a single building block, a smooth step $\sigma(\tau;\delta)$ that rises from 0 to 1 as τ crosses zero, with δ controlling the width or duration of the transition. As $\delta \to 0$, a Heaviside step is recovered. For finite $\delta > 0$, $\sigma(\tau)$ is continuously differentiable, so the transition can be resolved by a variable-step solver through step-size control rather than event detection.

Five implementations of σ are provided, allowing the user to trade the cost of numerically evaluating transcendental functions against the sharpness of the transition:

- `:ideal`: the Heaviside step,
- `:tanh`: $\sigma_{\tanh}(\tau;\delta) = \frac{1}{2}\big(1 + \tanh(\tau/\delta)\big)$,
- `:atan`: $\sigma_{\arctan}(\tau;\delta) = \frac{1}{2} + \frac{1}{\pi}\arctan(\tau/\delta)$,
- `:softmax` (logistic): $\sigma_{\log}(\tau;\delta) = \big(1 + e^{-\tau/\delta}\big)^{-1}$, for $\tau \geq 0$ and as $e^{\tau/\delta}/(1 + e^{\tau/\delta})$ otherwise, so that the exponential never overflows,
- `:algebraic`: $\sigma_{\text{alg}}(\tau;\delta) = \frac{1}{2}\big(1 + \tau/\sqrt{\tau^2 + \delta^2}\big)$ (free of transcendental functions and the default for PWM generators).

Periodic Carriers: PWM carriers are assembled from the selected implementation of σ and a periodic smooth triangular function. With $\varphi(t) \coloneqq \pi f_{\text{sw}} t$, a periodic triangular function at half the switching frequency is

$$T(t;\delta) \coloneqq 1 - \tfrac{2}{\pi}\arccos\big(-(1-\delta)\cos\varphi(t)\big),$$

where the factor $1-\delta$ keeps the argument of the $\arccos$ away from ± 1 and rounds the corners. A triangular carrier (at f_{sw}) is obtained by evaluating T directly and gating it at the start time, i.e.,

$$c(t) = T(2\varphi(t);\delta)\,\sigma(t - t_{\text{start}};\delta).$$

A smooth square signal at $f_{\text{sw}}/2$ is

$$Q(t;\delta) \coloneqq 2\sigma(\sin\varphi(t);\delta) - 1.$$

A product of $T(t;\delta)$ and $Q(t;\delta)$ has double the frequency, so the sawtooth carriers are

$$c(t) = \begin{cases} \frac{1}{2}\big(1 + T(t;\delta)\,Q(t;\delta)\big), & \text{sawtooth (rising)}, \\ \frac{1}{2}\big(1 - T(t;\delta)\,Q(t;\delta)\big), & \text{reversed sawtooth (falling)}. \end{cases}$$

both multiplied by the same start gate $\sigma(t - t_{\text{start}};\delta)$ as the triangular carrier. The PWM output is then

$$u_{\text{pwm}}(t) = q(t) = \sigma\Big(d(t) - c(t);\,\delta\Big)\,\sigma\Big(t - t_{\text{start}};\,\delta/f_{\text{sw}}\Big), \quad (2)$$

where $d(t)$ is the duty cycle and the second factor is a start gate of width δ/f_{sw}, a fixed fraction of the switching period.

IV. Benchmark Converters and Model Assembly

As an example of the library's capabilities, two benchmark converters are presented next, both under open-loop operation. The buck converter is a textbook case [9, §9.5.3] and serves to check correctness and solve time against a reference, whereas the full-bridge inverter is configured with a lightly damped output filter to act as a numerically demanding test case. Both are also implemented in MATLAB (2025b) with the Simscape Electrical toolbox for software-to-software cross-validation.

A. Buck Converter

Figure 2 illustrates the buck converter example taken from [9, §9.5.3]. The converter operates in continuous conduction mode (CCM), with an expected output of $v_o^* = \langle d(t)\rangle_T\, V_s = 15\,\text{V}$ and the theoretical inductor ripple is $\Delta i_L = V_s\,(1 - D)/2\,L\,f_{\text{sw}} = 1.3\,\text{A}$.

The converter is simulated with an input voltage $V_s = 28\,\text{V}$ and a switching frequency $f_{\text{sw}} = 100\,\text{kHz}$, with a steady-state duty cycle of 0.536. The output filter consists of an inductance $L = 50\,\mu\text{H}$ with a coil resistance $R_L = 10\,\text{m}\Omega$, and a capacitance $C = 500\,\mu\text{F}$. The load is purely resistive, with $R = 3\,\Omega$. The system is simulated over $400\,\text{ms}$.

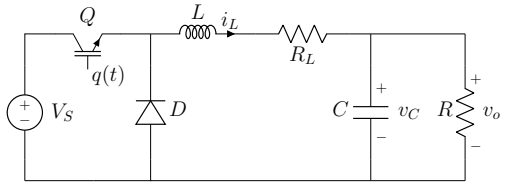


Fig. 2. Open-loop buck converter example.

We break down the model assembly with `MTKPowerConverters.jl` into three steps: (i) symbolic model creation, (ii) symbolic model "compilation",

[3]Cellier and Kofman [1, §9.1] attribute this to the fact that numerical integrators approximate trajectories by polynomials in the step size, which cannot represent a discontinuity. The solver experiences the discontinuity as a singular point of infinite stiffness and reduces its step size to the smallest tolerable value, which can degenerate into what they call "creeping behavior" along the switching boundary [1, §9.12].

and (iii) numerical model instantiation. The creation of the symbolic model is illustrated in Snippet II. Following an object-oriented approach, each component is instantiated from a prespecified model template (e.g., `PiecewiseLinearDiode` for the diode) with the appropriate parameter values (overwriting default values in any model template). The circuit is then assembled by connecting the components through their ports, and a symbolic system is created from the resulting connection list.

SNIPPET II
BUCK CONVERTER MODEL ASSEMBLY (EXCERPT).

```
@named vsource = Constant(k = 28.0)
@named source = Voltage()
@named inductor = LeakyInductor(L = 50e-6, R_series
    = 0.01, G_parallel = 1e-9)
@named switch = SwitchRCSnubber()
@named diode = PiecewiseLinearDiode()
@named capacitor = Capacitor(C = 500e-6)
@named load = Resistor(R = 3.0)
@named gnd = Ground()
% PWM
f_sw = 100e3;
@named pwm = PWMSawtooth(frequency = f_sw,
    start_time = 1e-9,
    delta = 1e-5, waveform = :algebraic)
@named duty_cycle = Constant(k = 0.536)

# Model connections
connections = [
    connect(vsource.output, source.V),
    connect(source.p, switch.p),
    connect(switch.n, diode.n),
    connect(diode.n, inductor.p),
    connect(inductor.n, capacitor.p),
    connect(capacitor.p, load.p),
    connect(source.n, diode.p, capacitor.n, load.n,
        gnd.g),
    connect(pwm.output, switch.ctrl),
    connect(duty_cycle.output, pwm.ref)
]

# Symbolic model
systems = [ vsource, source, inductor, switch,
    diode, capacitor, load, gnd, pwm, duty_cycle]

@named model = System(connections, t; systems =
    systems)
```

The second step is symbolic compilation, `mtkcompile()` in `ModelingToolkit.jl` terms, which expands the connections and reduces the acausal description to the causal model described in Section II, selecting the unknowns to be integrated and treating the remaining variables as observed quantities.

SNIPPET III
BUCK CONVERTER MODEL COMPILATION (EXCERPT).

```
buck_model_compiled = mtkcompile(buck_model;
    conservative = true, additional_passes = [
    IfLifting])
```

After symbolic compilation, the model remains unsuitable for numerical simulation because all variables are symbolic. The final step is to create a numerically efficient object, referred to in the SciML ecosystem as a `Problem`. From the object-oriented perspective, the `Problem` is a particular instance of the symbolic model. More concretely, if our converter is described by the ODE system $\dot{\boldsymbol{x}} = \mathcal{F}(\boldsymbol{x}; \boldsymbol{\theta})$, then an ODE problem corresponds to $\dot{\boldsymbol{x}} = \mathcal{F}(\boldsymbol{x}; \boldsymbol{\theta}_0)$ subject to $\boldsymbol{x}(0) = \boldsymbol{x}_0$. This notation emphasizes the two pieces of information that must be passed to create a problem: the parameter vector $\boldsymbol{\theta}_0$ and the initial state $\boldsymbol{x}_0$. The parameter vector might be obtained from the symbolic model, but it can also be modified to perform different simulations.

SNIPPET IV
BUCK CONVERTER NUMERICAL PROBLEM INSTANCE CREATION (EXCERPT).

```
prob = ODEProblem(
    buck_model_compiled,
    x0,
    (0.0, 400e-3),
    guesses=guesses,
    missing_guess_value=MTKBase.MissingGuessValue.
        Constant(0.0),
    warn_initialize_determined=false)
```

After symbolic compilation the system reduces to a 2-state ODE in (i_L, v_C), integrated with `Rodas5P` at tolerances $r_{\text{tol}} = a_{\text{tol}} = 10^{-6}$.

B. Full-Bridge Inverter

The inverter shown in Fig. 3 consists of four semiconductor devices, $\{Q_i\}_{i=1}^{4}$, arranged in two switching legs with an LC filter coupled at the output.

For simulation, the DC input voltage is set at $V_s = 400\,\text{V}$. The switches are driven using bipolar sinusoidal PWM with a triangular carrier and at a switching frequency $f_{\text{sw}} = 10\,\text{kHz}$. The sinusoidal modulating signal has the fundamental frequency $f_1 = 60\,\text{Hz}$, with a constant modulation index $m_a = 1.0$. The output filter consists of an inductance $L = 4.06\,\text{mH}$ with a coil resistance $R_L = 10\,\text{m}\Omega$, and a capacitance $C = 100\,\mu\text{F}$. The load is purely resistive, with $R = 50\,\Omega$. Unlike the buck converter, the filter values are chosen deliberately to give a lightly damped resonance ($f_0 = (2\pi\sqrt{LC})^{-1} \approx 250\,\text{Hz}$, damping ratio $\zeta \approx 0.06$) close to the low-order harmonics of f_1. The full-bridge inverter is implemented following the same approach presented in Section IV-A.

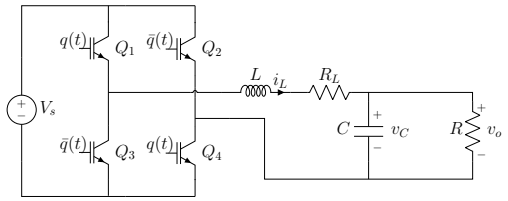


Fig. 3. Open-loop single-phase full-bridge inverter example.

In contrast to the constant duty cycle of the buck converter, the gating signals $q(t)$ and $\bar{q}(t) = 1 - q(t)$ of the two legs are obtained by comparing the triangular carrier at f_{sw} with the sinusoidal modulating signal at f_1.

V. Numerical Experiments

This section presents the comparison of PWM smoothing strategies that fixes the carrier implementation, followed by the validation of the two benchmark converters against Simscape.

A. PWM Implementation Comparison

Given the five implementations of σ in Section III-B, the question is how the smooth strategies compare with the ideal discontinuous option in approximation accuracy and computational cost, and which offers the best trade-off for typical use cases.

To this end, a sawtooth PWM generator is configured at $f_{\text{sw}} = 100\,\text{kHz}$ and duty cycle $\langle d(t)\rangle_T = 0.6$. The output $u_{\text{pwm}}(t)$ is computed once with `:ideal` (reference), and once with each of smooth implementations of σ in Section III-B. The differences in $u_{\text{pwm}}(t)$ are due to smoothing alone.

To evaluate the approximation error, the dimensionless parameter $\bar{\delta} = \delta/T_{\text{sw}}$, the smoothing width normalized by the switching period, is swept over $[0.001, 1]$, so that the results are comparable across switching frequencies.

For each value, the PWM output produced by each smooth implementation is compared with the `:ideal` output using the normalized root-mean-square error (NRMSE) over a steady-state window of length ΔT:

$$\text{NRMSE} = \frac{\sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(u_i - \tilde{u}_i\right)^2}}{\max_i u_i - \min_i u_i}, \tag{3}$$

where $u_i := u_{\text{pwm}}(t_i)$ is the ith of the N samples of the `:ideal` reference within ΔT and $\tilde{u}_i$ is the corresponding sample of the smooth approximation (for the unit-amplitude PWM output considered here the range is 1, so the NRMSE coincides with the root-mean-square error).

In addition to the NRMSE, the median solve time is reported to quantify the computational cost associated with each strategy. Before timing, each configuration is first executed at the largest value of $\bar{\delta}$ to trigger just-in-time (JIT) compilation and exclude compilation overhead from the measurements. Each configuration is subsequently evaluated using repeated solves, from which the median solve time is obtained. This procedure provides an estimate of the solver cost as a function of both $\bar{\delta}$ and the selected waveform strategy. The NRMSE, together with the median solve time t_{solve}, for each approximation is presented in Fig. 4.

As expected, the error of every smooth implementation vanishes as $\bar{\delta} \to 0$ and grows with $\bar{\delta}$ (Fig. 4, top). For small $\bar{\delta}$ it stays below $7\,\%$ for all implementations, with `tanh` and `softmax` the lowest. Both keep an approximately constant error up to $\bar{\delta} \approx 0.1$, beyond which it increases rapidly, whereas the `algebraic` NRMSE increases monotonically over the whole range, with a change of slope around $\bar{\delta} \approx 0.1$. Among the four, `atan` yields the largest error throughout.

In solve time (Fig. 4, bottom), `algebraic` is generally the fastest, as expected from its formulation free of transcendental functions, while `tanh`, the most accurate at larger $\bar{\delta}$, runs within the same order of time as the other transcendental implementations.

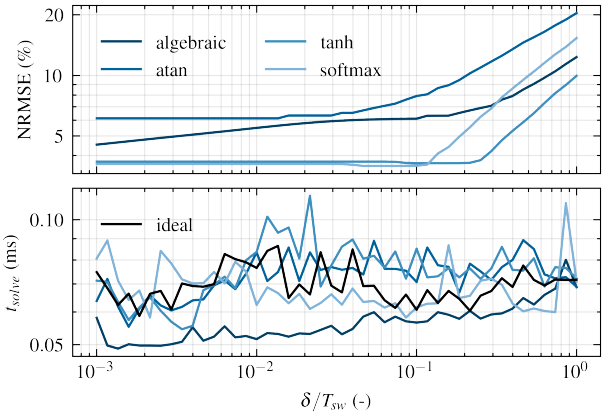


Fig. 4. NRMSE (top) and median solve time (bottom) of PWM waveform approximations vs. normalized smoothing width $\bar{\delta} = \delta/T_{\text{sw}}$, with $f_{\text{sw}} = 100\,\text{kHz}$ and duty cycle of $\langle d(t)\rangle_T = 0.6$ relative to the `:ideal` PWM.

Beyond accuracy and computational cost for a given $\bar{\delta}$, the choice of smoothing should also account for compatibility with the solver family in use and for its effect on the rest of the model, since $u_{\text{pwm}}(t)$ is the gate signal of a switch and its smoothing propagates to the switching transitions, the coupled converter dynamics, and the overall simulation time, none of which the generator-only sweep captures. For the benchmark converters that follow, `algebraic` appears to offer the best trade-off and is adopted as the default.

B. Buck Converter Validation

The Simscape implementation of the buck converter of Section IV-A, simulated with `ode23s`, serves as the reference. Figure 5 compares the steady-state waveforms.

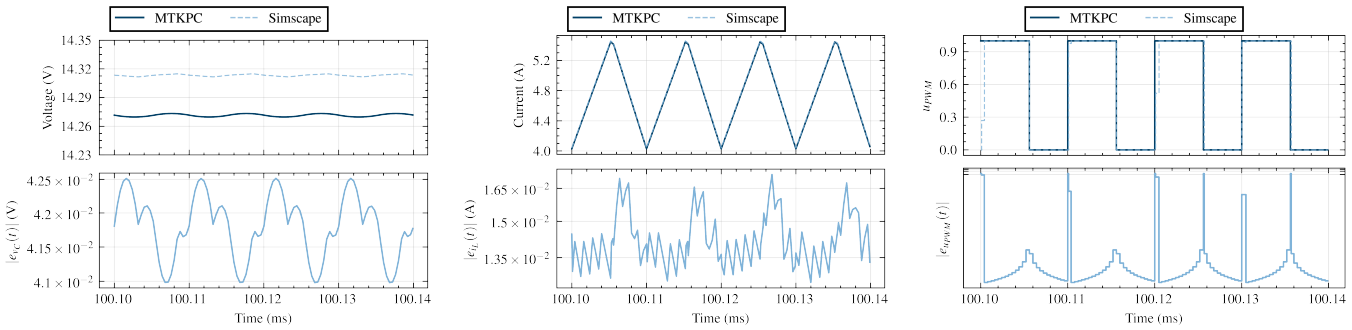
The output voltage settles to $V_o \approx 14.27\,\text{V}$, $4.86\,\%$ below v_o^*, which is consistent with the ohmic losses in R_L and R_{on}. The current ripple is close to the theoretical $1.3\,\text{A}$. The inductor current error e_{i_L} (Fig. 5b) is bounded by $\pm 0.2\,\text{A}$ and limited to the switching instants, and the voltage error e_{v_C} (Fig. 5a) is below $50\,\text{mV}$, so the two implementations agree in steady state with no constant offset between them.

Table I compares wall-clock timings for the same converter under the three solver configurations. Julia compilation includes symbolic simplification (`mtkcompile`) and `ODEProblem` instantiation, whereas MATLAB compilation is Simscape's model-compilation phase. End to end, the Julia run is 1.5 times faster than `ode23s` and 3.2 times faster than `daessc`, with compilation accounting for less than $0.4\,\text{s}$.

C. Inverter Validation

The inverter of Section IV-B is validated in the same way. Figure 6 compares the steady-state voltage and current waveforms and their harmonic content.

The output voltage is a sinusoid at $f_1 = 60\,\text{Hz}$ with a peak of about $424\,\text{V}$, $6\,\%$ above the $400\,\text{V}$ fundamental expected



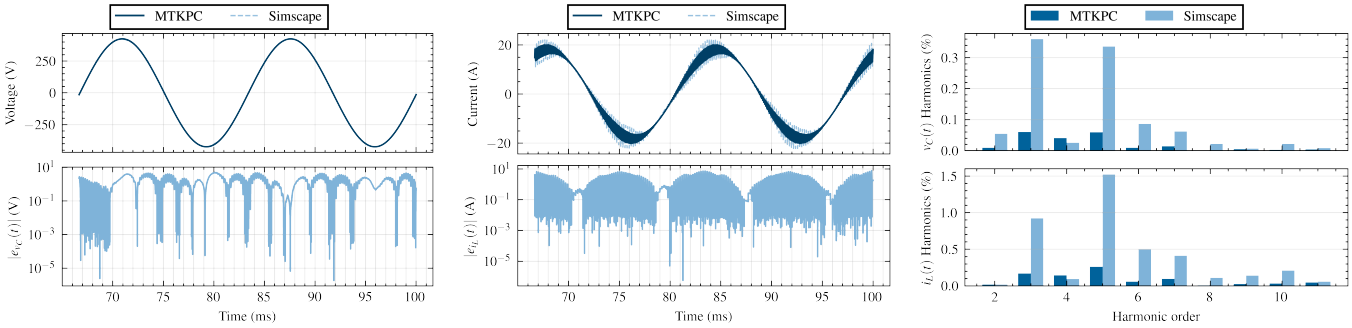
(a) Output voltage. (b) Inductor current. (c) PWM output.

Fig. 5. Steady-state Buck Converter waveform comparison: `MTKPowerConverters.jl` vs. Simscape (`ode23s`).

TABLE I
SIMULATION PERFORMANCE BENCHMARK ($n = 10$ RUNS, MEAN $\pm$ STD).

Phase	**Sub-phase**	**Julia** (`Rodas5P`)	**MATLAB** (`ode23s`)	**MATLAB** (`daessc`)
Compilation	Symbolic (`mtkcompile`)	$0.039 \pm 0.000\,\mathrm{s}$	$0.557 \pm 0.041\,\mathrm{s}$	$0.577 \pm 0.111\,\mathrm{s}$
	Numeric (`ODEProblem`)	$0.023 \pm 0.000\,\mathrm{s}$		
	Total	$0.400 \pm 0.137\,\mathrm{s}$		
Simulation		$6.797 \pm 0.091\,\mathrm{s}$	$10.472 \pm 0.113\,\mathrm{s}$	$23.067 \pm 0.265\,\mathrm{s}$
Total		$7.354 \pm 0.307\,\mathrm{s}$	$11.039 \pm 0.161\,\mathrm{s}$	$23.655 \pm 0.343\,\mathrm{s}$

Julia sub-phases were each measured independently using `BenchmarkTools.jl` over 10 runs; the compilation total time reports the end-to-end wall time for both sub-phases running sequentially. Julia values are truncated to three decimal places.



(a) Capacitor voltage $v_C(t)$. (b) Inductor current $i_L(t)$. (c) Harmonic content for $v_C(t)$ and $i_L(t)$

Fig. 6. Steady-state single-phase full-bridge inverter waveform comparison: `MTKPowerConverters.jl` vs. Simscape (`ode23s`).

for $m_a = 1$. The overshoot is due to the lightly damped LC filter, whose transfer function

$$G_f(s) = \frac{R}{LCRs^2 + (L + R_L RC)s + (R_L + R)}$$

evaluated at f_1 gives a peak of $423.2\,\mathrm{V}$, in close agreement with the simulation.

As intended, the light damping makes this case a useful stress test, since it raises the sensitivity to resonant and harmonic excitation. Even so, the output voltage agrees with the Simscape reference to within $2\,\%$ of the $400\,\mathrm{V}$ peak.

The larger distortion of the inductor current is also consistent with the underdamped filter. The filter amplifies components near f_0 and responds to the switching transitions at $f_{\mathrm{sw}} = 10\,\mathrm{kHz}$, and since i_L contains the capacitor current $C\,\dot{v}_C$, these components are more pronounced in the current than in the voltage. With identical circuit parameters, the Simscape model shows a higher current distortion, which may stem from the different treatment of switching events by `Rodas5P` and `ode23s` and from the PWM generators.

The harmonic content in Fig. 6c is obtained from the discrete fourier transform (DFT) of one fundamental period in steady state. The fundamental is the spectral peak around f_1, harmonics up to the 11th are read at the bins nearest to the integer multiples of f_1, and each amplitude is expressed as a percentage of the fundamental, with the total harmonic distortion (THD) taken as the root-sum-square of the harmonics over the fundamental. The same procedure is

applied to v_C and i_L in both simulation environments.

The 3rd (180 Hz) and 5th (300 Hz) harmonics, closest to f_0, are the most affected by the resonance. The resulting THD values for voltage and current are $0.096\,\%$ and $0.368\,\%$ for `MTKPowerConverters.jl`, against $0.509\,\%$ and $1.920\,\%$ for the Simscape reference, so the library shows the lower distortion in this lightly damped case. Further investigation is needed to investigate where this discrepancy arises.

VI. Potential Integration with the SciML Ecosystem

The parameter vector $\boldsymbol{\theta}$ remains symbolically indexed after compilation, and the compiled problem is differentiable with the Julia AD toolchain [6], which enables workflows beyond simulation:

- **Parameter identification:** differentiate through the ODE solver to fit R_{on}, L, C to measured waveforms by gradient-based optimization.
- **Physics-informed ML:** for large-scale systems, models of selected components (e.g., the converters within a solar or wind farm) can be replaced by neural networks trained jointly with the physical ODE.
- **Multi-scale integration:** loss and impedance parameters extracted from the switching-level models can be passed to the averaged state-space representations used by grid-level tools such as `ElectricGrid.jl` [8], bridging the circuit and grid simulation scales.

VII. Conclusion and Future Work

This paper presented `MTKPowerConverters.jl`, a proof-of-concept Julia library in which every converter component is specified by its governing equations and compiled by `ModelingToolkit.jl`, so that the same transparency the Modelica community regards as foundational for reproducible simulation is brought to power electronics applications without the opacity of binary-compiled tools. The concept was tested in three numerical experiments. Smooth surrogates of the PWM waveforms, built from a single parametric step, give a controlled trade-off between accuracy and solve time, with the `algebraic` implementation as the balanced default. For a textbook buck converter, the steady-state waveforms agree with Simscape to within the switching ripple, and the end-to-end run is 1.5 to 3.2 times faster depending on the MATLAB solver. For a lightly damped inverter chosen as a numerically stressing case, the library reproduces the resonant overshoot predicted by the filter transfer function and shows lower harmonic distortion than the reference. Because the models are SciML native, they are directly usable in differentiable and physics-informed workflows.

Future work will proceed in two directions. On the modeling side, average-value models of the same converters will complement the switching-level ones [2], [9], so that steady-state analysis and control-loop design can be carried out on the averaged dynamics and verified on the switching model without changing the circuit description. On the scope side, the catalog will grow toward multi-converter and grid-connected topologies, converging on the grid-level abstraction of tools such as `ElectricGrid.jl` (Section VI) while keeping full equation transparency at every level.

References

[1] F. E. Cellier and E. Kofman, *Continuous System Simulation*. New York: Springer, 2006.

[2] H. Bai, C. Liu, D. Majstorovic, and F. Gao, *Real-Time Simulation Technology for Modern Power Electronics*. Elsevier, 2023.

[3] B. P. Zeigler, A. Muzy, and E. Kofman, *Theory of Modeling and Simulation: Discrete Event and Iterative System Computational Foundations*, 3rd ed. San Diego, CA: Academic Press, 2019.

[4] P. A. Fritzson, *Principles of Object-oriented Modeling and Simulation with Modelica 3.3 - A Cyber-Physical Approach*, 2nd ed. Wiley IEEE Press, 2015.

[5] A. M. G. Elsheikh and P. Palensky, *Modelica by Application: Power Systems*, 2021.

[6] Y. Ma, S. Gowda, R. Anantharaman, C. Laughman, V. Shah, and C. Rackauckas, "ModelingToolkit: A Composable Graph Transformation System For Equation-Based Modeling," Feb. 2022.

[7] C. Rackauckas and Q. Nie, "DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia," *Journal of Open Research Software*, vol. 5, no. 1, May 2017.

[8] O. Wallscheid, S. Peitz, J. Stenner, D. Weber, S. Boshoff, M. Meyer, V. Chidananda, and O. Schweins, "ElectricGrid.jl - A Julia-based modeling and simulation tool for power electronics-driven electric energy grids," *Journal of Open Source Software*, vol. 8, no. 89, p. 5616, Sep. 2023.

[9] R. W. Erickson and D. Maksimović, *Fundamentals of Power Electronics*, 3rd ed. Springer International Publishing, 2020.

[10] S. M. Tripathi and F. M. Gonzalez-Longatt, Eds., *Real-Time Simulation and Hardware-in-the-Loop Testing Using Typhoon HIL*, ser. Transactions on Computer Systems and Networks. Singapore: Springer Nature Singapore, 2023.